Analysis and Visualization Algorithms in VMD

David Hardy
http://www.ks.uiuc.edu/Research/~dhardy/
NAIS: State-of-the-Art Algorithms for Molecular Dynamics
(Presenting the work of John Stone.)
VMD – “Visual Molecular Dynamics”

• Visualization and analysis of molecular dynamics simulations, sequence data, volumetric data, quantum chemistry simulations, particle systems, …
• User extensible with scripting and plugins
• http://www.ks.uiuc.edu/Research/vmd/
### GPU Accelerated Trajectory Analysis and Visualization in VMD

<table>
<thead>
<tr>
<th>GPU-Accelerated Feature</th>
<th>GPU Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Molecular orbital display</td>
<td>120x</td>
</tr>
<tr>
<td>Radial distribution function</td>
<td>92x</td>
</tr>
<tr>
<td>Electrostatic field calculation</td>
<td>44x</td>
</tr>
<tr>
<td>Molecular surface display</td>
<td>40x</td>
</tr>
<tr>
<td>Ion placement</td>
<td>26x</td>
</tr>
<tr>
<td>MDFF density map synthesis</td>
<td>26x</td>
</tr>
<tr>
<td>Implicit ligand sampling</td>
<td>25x</td>
</tr>
<tr>
<td>Root mean squared fluctuation</td>
<td>25x</td>
</tr>
<tr>
<td>Radius of gyration</td>
<td>21x</td>
</tr>
<tr>
<td>Close contact determination</td>
<td>20x</td>
</tr>
<tr>
<td>Dipole moment calculation</td>
<td>15x</td>
</tr>
</tbody>
</table>

![Graph showing radial distribution function](image-url)
VMD for Demanding Analysis Tasks
Parallel VMD Analysis w/ MPI

- Analyze trajectory frames, structures, or sequences in parallel on clusters and supercomputers:
  - Compute time-averaged electrostatic fields, MDFF quality-of-fit, etc.
  - Parallel rendering, movie making
- Addresses computing requirements beyond desktop
- User-defined parallel reduction operations, data types
- Dynamic load balancing:
  - Tested with up to 15,360 CPU cores
- Supports GPU-accelerated clusters and supercomputers
Time-Averaged Electrostatics Analysis on Energy-Efficient GPU Cluster

• **1.5 hour** job (CPUs) reduced to **3 min** (CPUs+GPU)
• Electrostatics of thousands of trajectory frames averaged
• Per-node power consumption on NCSA “AC” GPU cluster:
  – CPUs-only: 299 watts
  – CPUs+GPUs: 742 watts
• GPU Speedup: **25.5x**
• Power efficiency gain: **10.5x**

Time-Averaged Electrostatics Analysis on NCSA Blue Waters Early Science System

<table>
<thead>
<tr>
<th>NCSA Blue Waters Node Type</th>
<th>Seconds per trajectory frame for one compute node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray XE6 Compute Node: 32 CPU cores (2xAMD 6200 CPUs)</td>
<td>9.33</td>
</tr>
<tr>
<td>Cray XK6 GPU-accelerated Compute Node: 16 CPU cores + NVIDIA X2090 (Fermi) GPU</td>
<td>2.25</td>
</tr>
</tbody>
</table>

Speedup for GPU XK6 nodes vs. CPU XE6 nodes

GPU nodes are 4.15x faster overall

Preliminary performance for VMD time-averaged electrostatics w/ Multilevel Summation Method running Blue Waters Early Science System
Visualizing Molecular Orbitals

• Visualization of MOs aids in understanding the chemistry of molecular system
• Display of MOs can require tens to hundreds of seconds on multi-core CPUs, even with hand-coded SSE
• GPUs enable MOs to be computed and displayed in a fraction of a second, fully interactively
MO GPU Parallel Decomposition

MO 3-D lattice decomposes into 2-D slices (CUDA grids)

Small 8x8 thread blocks afford large per-thread register count, shared memory

Each thread computes one MO lattice point.

Padding optimizes global memory performance, guaranteeing coalesced global memory accesses

Grid of thread blocks

Lattice can be computed using multiple GPUs

Threads producing results that are used

Threads producing results that are discarded

GPU 0

GPU 1

GPU 2

0,0 0,1 ...

1,0 1,1 ...

... ... ...

BTRC for Macromolecular Modeling and Bioinformatics
http://www.ks.uiuc.edu/

Beckman Institute, UIUC
VMD MO GPU Kernel Snippet:
Loading Tiles Into Shared Memory On-Demand

[... outer loop over atoms ...]
if ((prim_counter + (maxprim<<1)) >= SHAREDSIZE) {
  prim_counter += sblock_prim_counter;
  sblock_prim_counter = prim_counter & MEMCOAMASK;
  s_basis_array[sidx] = basis_array[sblock_prim_counter + sidx];
  s_basis_array[sidx + 64] = basis_array[sblock_prim_counter + sidx + 64];
  s_basis_array[sidx + 128] = basis_array[sblock_prim_counter + sidx + 128];
  s_basis_array[sidx + 192] = basis_array[sblock_prim_counter + sidx + 192];
  prim_counter -= sblock_prim_counter;
  __syncthreads();
}
for (prim=0; prim < maxprim; prim++) {
  float exponent = s_basis_array[prim_counter];
  float contract_coeff = s_basis_array[prim_counter + 1];
  contracted_gto += contract_coeff * __expf(-exponent*dist2);
  prim_counter += 2;
}[... continue on to angular momenta loop ...]

Shared memory tiles:

• Tiles are checked and loaded, if necessary, immediately prior to entering key arithmetic loops

• Adds additional control overhead to loops, even with optimized implementation
VMD MO GPU Kernel Snippet: 
Fermi kernel based on L1 cache

[... outer loop over atoms ...]
// loop over the shells belonging to this atom (or basis function)
for (shell=0; shell < maxshell; shell++) {
    float contracted_gto = 0.0f;
    int maxprim = shellinfo[(shell_counter<<4)];
    int shell_type = shellinfo[(shell_counter<<4) + 1];
    for (prim=0; prim < maxprim; prim++) {
        float exponent = basis_array[prim_counter];
        float contract_coeff = basis_array[prim_counter + 1];
        contracted_gto += contract_coeff * __expf(-exponent*dist2);
        prim_counter += 2;
    }
[... continue on to angular momenta loop ...]

L1 cache:
• Simplifies code!
• Reduces control overhead
• Gracefully handles arbitrary-sized problems
• Matches performance of constant memory
### VMD Single-GPU Molecular Orbital Performance Results for $C_{60}$

Intel X5550 CPU, GeForce GTX 480 GPU

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Cores/GPUs</th>
<th>Runtime (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon 5550 ICC-SSE</td>
<td>1</td>
<td>30.64</td>
<td>1.0</td>
</tr>
<tr>
<td>Xeon 5550 ICC-SSE</td>
<td>8</td>
<td>4.13</td>
<td>7.4</td>
</tr>
<tr>
<td>CUDA shared mem</td>
<td>1</td>
<td>0.37</td>
<td>83</td>
</tr>
<tr>
<td>CUDA L1-cache (16KB)</td>
<td>1</td>
<td>0.27</td>
<td>113</td>
</tr>
<tr>
<td>CUDA const-cache</td>
<td>1</td>
<td>0.26</td>
<td>117</td>
</tr>
<tr>
<td>CUDA const-cache, zero-copy</td>
<td>1</td>
<td>0.25</td>
<td>122</td>
</tr>
</tbody>
</table>

Fermi GPUs have caches: may outperform hand-coded shared memory kernels. Zero-copy memory transfers improve overlap of computation and host-GPU I/Os.
VMD Multi-GPU Molecular Orbital Performance Results for C_{60}

Intel X5550 CPU, 4x GeForce GTX 480 GPUs,

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Cores/GPUs</th>
<th>Runtime (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel X5550-SSE</td>
<td>1</td>
<td>30.64</td>
<td>1.0</td>
</tr>
<tr>
<td>Intel X5550-SSE</td>
<td>8</td>
<td>4.13</td>
<td>7.4</td>
</tr>
<tr>
<td>GeForce GTX 480</td>
<td>1</td>
<td>0.255</td>
<td>120</td>
</tr>
<tr>
<td>GeForce GTX 480</td>
<td>2</td>
<td>0.136</td>
<td>225</td>
</tr>
<tr>
<td>GeForce GTX 480</td>
<td>3</td>
<td>0.098</td>
<td>312</td>
</tr>
<tr>
<td>GeForce GTX 480</td>
<td>4</td>
<td>0.081</td>
<td>378</td>
</tr>
</tbody>
</table>

Uses persistent thread pool to avoid GPU init overhead, dynamic scheduler distributes work to GPUs
Molecular Orbital Computation and Display Process

For each trajectory frame, for each MO shown:

1. **Read QM simulation log file, trajectory**
2. **Preprocess MO coefficient data**
   - eliminate duplicates, sort by type, etc...
3. **For current frame and MO index, retrieve MO wavefunction coefficients**
4. **Compute 3-D grid of MO wavefunction amplitudes**
5. **Most performance-demanding step, run on GPU...**
6. **Extract isosurface mesh from 3-D MO grid and render the resulting surface**
7. **Apply user coloring/texturing**

One-time initialization:
- **Initialize Pool of GPU Worker Threads**

One-time initialization:
- **BTRC for Macromolecular Modeling and Bioinformatics**
  - http://www.ks.uiuc.edu/BIRC

Beckman Institute, UIUC
Multi-GPU Load Balance

- Many early CUDA codes assumed all GPUs were identical
- Host machines may contain a diversity of GPUs of varying capability (discrete, IGP, etc)
- Different GPU on-chip and global memory capacities may need different problem “tile” sizes
- Static decomposition works poorly for non-uniform workload, or diverse GPUs
Multi-GPU Dynamic Work Distribution

// Each GPU worker thread loops over
// subset 2-D planes in a 3-D cube…
while (!threadpool_next_tile(&parms,
tilesiz, &tile)){
  // Process one plane of work…
  // Launch one CUDA kernel for each
  // loop iteration taken…
  // Shared iterator automatically
  // balances load on GPUs
}

Dynamic work distribution
Example Multi-GPU Latencies Relevant to Interactive Sci-Viz, Script-Driven Analyses
(4 Tesla C2050 GPUs, Intel Xeon 5550)

6.3us    CUDA empty kernel (immediate return)
9.0us    Sleeping barrier primitive (non-spinning barrier that uses POSIX condition variables to prevent idle CPU consumption while workers wait at the barrier)
14.8us   pool wake, host fctn exec, sleep cycle (no CUDA)
30.6us   pool wake, 1x(tile fetch, simple CUDA kernel launch), sleep
1817.0us pool wake, 100x(tile fetch, simple CUDA kernel launch), sleep
## Multi-GPU Dynamic Scheduling Performance with Heterogeneous GPUs

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Cores/GPUs</th>
<th>Runtime (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel X5550-SSE</td>
<td>1</td>
<td>30.64</td>
<td>1.0</td>
</tr>
<tr>
<td>Quadro 5800</td>
<td>1</td>
<td>0.384</td>
<td>79</td>
</tr>
<tr>
<td>Tesla C2050</td>
<td>1</td>
<td>0.325</td>
<td>94</td>
</tr>
<tr>
<td>GeForce GTX 480</td>
<td>1</td>
<td>0.255</td>
<td>120</td>
</tr>
<tr>
<td>GeForce GTX 480 +</td>
<td>3</td>
<td>0.114</td>
<td>268</td>
</tr>
<tr>
<td>Tesla C2050 + Quadro 5800</td>
<td></td>
<td></td>
<td>(91% of ideal perf)</td>
</tr>
</tbody>
</table>

Dynamic load balancing enables mixture of GPU generations, SM counts, and clock rates to perform well.
Multi-GPU Runtime Error/Exception Handling

- Competition for resources from other applications can cause runtime failures, e.g. GPU out of memory half way through an algorithm
- Handle exceptions, e.g. convergence failure, NaN result, insufficient compute capability/features
- Handle and/or reschedule failed tiles of work
Radial Distribution Functions

- RDFs describes how atom density varies with distance
- Can be compared with experiments
- Shape indicates phase of matter: sharp peaks appear for solids, smoother for liquids
- Quadratic time complexity $O(N^2)$
Computing RDFs

- Compute distances for all pairs of atoms between two groups of atoms A and B
- A and B may be the same, or different
- Use nearest image convention for periodic systems
- Each pair distance is inserted into a histogram
- Histogram is normalized one of several ways depending on use, but usually according to the volume of the spherical shells associated with each histogram bin
Multi-GPU RDF Performance

- 4 NVIDIA GTX480 GPUs 30 to 92x faster than 4-core Intel X5550 CPU
- Fermi GPUs ~3x faster than GT200 GPUs: larger on-chip shared memory

Molecular Surface Display: “QuickSurf” Representation

• Displays continuum of structural detail:
  – All-atom models
  – Coarse-grained models
  – Cellular scale models
  – Multi-scale models: All-atom + CG, Brownian + Whole Cell
  – Smoothly variable between full detail, and reduced resolution representations of very large complexes

• Uses multi-core CPUs and GPU acceleration to enable smooth real-time animation of MD trajectories

• Linear-time algorithm, scales to hundreds of millions of particles, as limited by memory capacity

Fast Visualization of Gaussian Density Surfaces for Molecular Dynamics and Particle System Trajectories.
M. Krone, J. Stone, T. Ertl, K. Schulten. EuroVis 2012. (Submitted)
Recurring Algorithm Design Principles (1)

• Extensive use of on-chip shared memory and constant memory to further amplify memory bandwidth
• Pre-processing and sorting of operands to organize computation for peak efficiency on the GPU, particularly for best use of L1 cache and shared mem
• Tiled/blocked data structures in GPU global memory for peak bandwidth utilization
• Use of CPU to “regularize” the work done by the GPU, handle exceptions & unusual work units
• Asynchronous operation of CPU/GPU enabling overlapping of computation and I/O on both ends
Recurring Algorithm Design Principles (2)

• Take advantage of special features of the GPU memory systems
  – Broadcasts, wide loads/stores (float4, double2), texture interpolation, write combining, etc.

• Avoid doing complex array indexing arithmetic within the GPU threads, pre-compute as much as possible outside of the GPU kernel so the GPU is doing what it’s best at: floating point arithmetic